Parking Lot

Nature of the Game

We want to understand how you think as a programmer, and the level of craft you bring to bear when building software.

Of course, the ideal would be a real world problem, with real scale, but that isn't practical as it would take too much time. So instead we have a dead simple, high school level problem that we want you to solve as though it was a real-world problem.

Please note that not following the instructions below will result in an automated rejection. Taking longer than the time allocated will negatively affect our evaluation of your submission.

Rules of the Game

1. You have two full days to implement a solution.
2. Please implement your solution with Golang version >= 1.15.3.
3. We are really, really interested in your object oriented or functional design skills, so please craft the most beautiful code you can.
4. We're also interested in understanding how you make assumptions when building software. If a particular workflow or boundary condition is not defined in the problem statement below, what you do is your choice.
5. You have to solve the problem in any object oriented or functional language without using any external libraries to the core language except for a HTTP router library (like go-chi/chi, gorilla/mux, etc.) and library for unit testing. Your solution can be built into a binary file.
6. Please use Git for version control. We expect you to send us a standard zip or tarball of your source code when you're done that includes Git metadata (the .git folder) in the tarball so we can look at your commit logs and understand how your solution evolved. Frequent commits are a huge plus.
7. Please do not commit binaries, class files, jars, libraries and output from the build process.
8. Please write comprehensive unit tests/specs. For object oriented solutions, it's a huge plus if you test drive your code. Submitting your solution with only a functional suite will result in a rejection.

9. Please create your solution inside the `parking_lot` directory. Your codebase should have the same level of structure and organization as any mature open source project including coding conventions, directory structure and build approach (Makefile, Dockerfile, etc) and a README.md with clear instructions.

10. Your submission must pass the automated test. In order to make sure that we can run your code correctly, please provide a script in `bin/setup` that would install dependencies and/or compile the code and then run your unit test suite. You also need to provide a script in `bin/parking_lot` to run the program itself. Please note that these files are Unix executable files and should run on Unix.

11. Please ensure that you follow the syntax and formatting of both the input and output samples. You can use the automated functional test included in the zip file we've sent to you. This is to help you validate the correctness of your program. You can run it by invoking `bin/functional_test`. **IMPORTANT**: To make the functional test work correctly, some setup is needed. Instructions to set up the functional suite can be found under README.md at the mentioned inside the zip file.

12. Please do not make either your solution or this problem statement publicly available by using public github or bitbucket or by posting this problem to a blog or forum.

Problem Statement

I own a parking lot that can hold up to 'n' cars at any given point in time. Each slot is given a number starting at 1 increasing with the increasing distance from the entry point in steps of one. I want to create an automated ticketing system in the cloud that allows my customers to use my parking lot without human intervention.

When a car enters my parking lot, I want to have a ticket issued to the driver. The ticket issuing process includes us documenting the registration number (number plate) and the colour of the car and allocating an available parking slot to the car before actually handing over a ticket to the driver (we assume that our customers are nice enough to always park in the slots allocated to them). The customer should be allocated to a parking slot which is nearest to the entry. At the exit the customer returns the ticket which then marks the slot they were using as being available.

Due to government regulation, the system should provide me with the ability to find out:

- Registration numbers of all cars of a particular colour.
- Slot number of a car with a given registration number is parked.
- Slot numbers of all slots where cars of a particular colour are parked.

We interact with the system via a set of endpoints which produce a specific output. Please take a look at the example below, which includes all the endpoints you need to support - they're self

explanatory. The system should allow input in two ways. Just to clarify, the same codebase should support both modes of input - we don't want two distinct submissions.

1. It should provide us with several HTTP APIs as the commands.
2. It should accept a POST HTTP Request that accepts plain text payload which contains all the commands and reads the commands from the payload.

Please note that the HTTP server should run on port `8080` in your local machine.

Example: Several HTTP APIs

To install all dependencies, compile and run tests:
`$ /bin/setup`

To run the program:
`$ /bin/parking_lot`

Assuming a parking lot with 6 slots, the following commands should be run in sequence by typing them in a tool like cURL and should produce output as described below the command.

```
$ curl -X POST localhost:8080/create_parking_lot/6
Created a parking lot with 6 slots

$ curl -X POST http://localhost:8080/park/B-1234-RFS/Black
Allocated slot number: 1

$ curl -X POST http://localhost:8080/park/B-1999-RFD/Green
Allocated slot number: 2

$ curl -X POST http://localhost:8080/park/B-1000-RFS/Black
Allocated slot number: 3

$ curl -X POST http://localhost:8080/park/B-1777-RFU/BlueSky
Allocated slot number: 4

$ curl -X POST http://localhost:8080/park/B-1701-RFL/Blue
Allocated slot number: 5

$ curl -X POST http://localhost:8080/park/B-1141-RFS/Black
Allocated slot number: 6

$ curl -X POST http://localhost:8080/leave/4
Slot number 4 is free

$ curl -X GET http://localhost:8080/status
Slot No. Registration No Colour
1 B-1234-RFS Black
2 B-1999-RFD Green
3 B-1000-RFS Black
5 B-1701-RFL Blue
6 B-1141-RFS Black
```

```
$ curl -X POST http://localhost:8080/park/B-1333-RFS/Black
Allocated slot number: 4

$ curl -X POST http://localhost:8080/park/B-1989-RFU/White
Sorry, parking lot is full

$ curl -X GET http://localhost:8080/cars_registration_numbers/colour/Black
B-1234-RFS, B-1000-RFS, B-1333-RFS, B-1141-RFS

$ curl -X GET http://localhost:8080/cars_slot/colour/Black
1, 3, 4, 6

$ curl -X GET http://localhost:8080/slot_number/car_registration_number/B-1701-RFL
5

$ curl -X GET http://localhost:8080/slot_number/car_registration_number/RI-1
Not found
```

Example: POST HTTP API with plain text payload

Path: `POST http://localhost:8080/bulk`

Request HTTP Body (text/plain):

```
create_parking_lot 6
park B-1234-RFS Black
park B-1999-RFD Green
park B-1000-RFS Black
park B-1777-RFU BlueSky
park B-1701-RFL Blue
park B-1141-RFS Black
leave 4
status
park B-1333-RFS Black
park B-1989-RFU BlueSky
registration_numbers_for_cars_with_colour Black
slot_numbers_for_cars_with_colour Black
slot_number_for_registration_number B-1701-RFL
slot_number_for_registration_number RI-1
```

Response HTTP Body (text/plain):

```
Created a parking lot with 6 slots
Allocated slot number: 1
Allocated slot number: 2
Allocated slot number: 3
Allocated slot number: 4
Allocated slot number: 5
Allocated slot number: 6
Slot number 4 is free
Slot No. Registration No Colour
1 B-1234-RFS Black
2 B-1999-RFD Green
```

```
3 B-1000-RFS Black
5 B-1701-RFL Blue
6 B-1141-RFS Black
Allocated slot number: 4
Sorry, parking lot is full
B-1234-RFS, B-1000-RFS, B-1333-RFS, B-1141-RFS
1, 3, 4, 6
5
Not found
```